Assignment 01

Reliable Data Transport Candidate: 153728

1 Stop-and-Wait

1.1 Sender

This stop-and-wait transport protocol is implemented using a finite-state machine, described in the `Sender` class by an enum `SenderState` containing the values `WAIT_MSG` and `WAIT_ACK`. In the `WAIT_MSG` state, the sender waits for a message from the application layer. In the `WAIT_ACK` state, a packet is currently 'in-transit'. That is, the sender is waiting for acknowledgement from the receiver. An integer field `seq` keeps track of the current packet. Since the stop-and-wait protocol is an alternating bit protocol, `seq` only holds either of the values 0 or 1 (this is done by always setting its value modulo 2). In the `WAIT_MSG` state this field is used for the sequence number of the next message from the application layer.

1.2 Receiver

The receiver initialises its state with an expected sequence number of `expectedSeq=0` (this represents the sequence number of the packet wanted from the sender). Then it awaits a packet from the network. Whenever a new packet is received, it is checked for corruption and for having the required sequence number. If the packet is either corrupt or has the wrong sequence number then the receiver acknowledges the previously received packet (0 if `expectedSeq` is 1, 1 if `expectedSeq` is 0). This causes the sender host to resend the expected packet. Otherwise, if the received packet is valid, the payload is delivered up to the application layer and then a response is sent to the sender host which acknowledges the received packet, with acknowledgement number equal to `expectedSeq`. Then `expectedSeq` is set to the alternate value $0 \to 1$, $1 \to 0$ and the receiver now waits for the next packet.

2 Go-Back-N

The Stop-and-Wait protocol is a specific case of the Go-Back-N protocol. Indeed, the Stop-and-Wait protocol can be recovered from setting the `WINDOW_SIZE` in the following implementation to 1.

2.1 Sender

Upon receiving a new message from the application layer, if there is free space in the buffer, a new packet is created and stored with the message data and a sequence number of `nextSeqNum`. Then if the sequence number is within the window range (`base` $\leq$ `nextSeqNum` $<$ `base` + `WINDOW_SIZE`), the newly created packet is sent to the client and if the packet is the first in the window, the timer is started. In both cases, `nextSeqNum` is then incremented.

Upon receiving a new packet from the other host, the packet is first checked to be corrupt - if it is, it is ignored and no further action is taken. Otherwise the acknowledgement number of the packet is extracted - all packets with a sequence number below or equal to this have been successfully received by the receiver. In the initial implementation, this means the base could be immediately shifted up to one more than the received acknowledgement number. However this implementation involves buffering, so the actions taken at this point had to be modified. If the packet's acknowledgement number is less than `base` then it is ignored, since it is acknowledging an already-acknowledged packet. Otherwise, the running timer is stopped because the actions taken at this point will update the state of the sender. Then the window is stepped forward one slot at a time, and at each step a check is performed to see if a buffered packet at the new 'end' of the window exists - if so, it is transmitted to the receiver. Finally, unless the window is now empty, the timer is started again - if the window is empty the timer should remain off because there are no in-flight packets, and if non-empty the timer should be started again to track the base packet.

Upon a timeout, all existing packets in the window are simply re-sent in order to the receiver and the timer is restarted.

In terms of the design of the sender implementation: the final design uses a single buffer for both storing the in-window packets and the buffered packets. The initial design instead used a separate queue of packets (`ArrayDeque<Packet>`) - upon a new acknowledgement packet being received and the window being moved, packets would be polled from the queue until either the queue was empty or the window was full. This was initially easier to implement, but eventually it was decided that having two separate data structures could make the implementation more bug-prone. In addition, the queue had the potential to grow unboundedly large if messages flooded in from the application layer resulting in running out of memory rather than dropping the messages. Even though this could have been prevented by keeping track of the queue size, in the end it was decided that a single array would be a less complex, neater, and less bug-prone solution.

2.2 Receiver

The receiver implementation is more straightforward than the sender. The sender has two fields: `nextSeqNum`, holding the sequence number of the next packet required from the sender, and `sendPacket`, holding a ready-to-send packet for acknowledging the packet with the highest in-order sequence number.

After the initial setup of the fields (with `nextSeqNum` set to 1), the receiver has only one state, which waits for an incoming packet. If an incoming packet is corrupt or does not have the expected sequence number, then `sendPacket` is transmitted through the network to notify the sender of the packets that have been successfully delivered. Otherwise, if the packet is valid the payload is delivered to the application layer, `sendPacket`'s acknowledgement number is set to the sequence number of the received packet and transmitted to the sender. Finally, `nextSeqNum` is incremented so that the receiver and the receiver now waits for the packet with this updated sequence number.

3 Other

In both protocols, the identical checksum calculation methods are used. In the `Checksum` class, there are methods for calculating the checksum of packet contents and for validating a packet's checksum. The technique used for calculating the checksum are similar to TCP: the sequence and acknowledgement numbers are added together, then the integer value of each character of the payload is added to this total. Finally, the bitwise NOT of the total (`~total`) is returned. This utility class also contains a convenience method for checking if a packet is corrupt, by calculating the checksum of the packet fields and comparing it to the value of the packet's checksum field.